Q94 : 二叉树的中序遍历 非递归

1. 若一个结点有左孩子,就一直把当前结点入栈.然后指向左孩子
2. 从栈中取出一个结点,访问, 然后若该结点有右孩子,就把右孩子入栈.若右孩子有左孩子,则同1.

Q144 : 二叉树的前序遍历 非递归

1. 首先访问root结点
2. 然后把当前结点的右结点放入stack, 然后把左孩子放入stack中。
3. 若栈不为空, 出栈一个，首先访问该结点，然后执行2

Q145 : 二叉树的后序遍历 ： 非递归

1. 如果按照访问路线来控制，那么该程序会很难编写。
2. 对前序遍历进行改造。 Q144中前序遍历的入栈是，先入右结点。 后序中反着来，先入左结点。通过逆序入栈顺序获取一个遍历序列。
    1. 前序遍历 1245367
    2. 后序遍历 4526731
    3. 逆序入栈的前序遍历 1376254
    4. 可以发现逆序入栈的前序遍历恰好是 后序遍历的逆序。所以对 3直接逆序输出即可。
    5. 该题返回的是 List ,那么可以使用linkedlist 使用头插法，直接得到逆序的结果。省下逆序的时间。
3. 附加mark结点法：虚拟： 在每个非叶子结点下附加一个mark结点，然后从mark结点分支出两个子节点。
    1. 当出栈结点是非mark结点时，这个时候需要入栈他的子结点了，但是，在入栈子结点时，先入栈一个mark结点。
    2. 当入栈非mark结点时，就是第一次访问。 当出栈结点是mark结点时，代表了对root结点的第二次访问。
    3. 当真正的root结点出栈时就是对root结点的第三次访问。
4. 传统的非递归法： 也是最难写的。 暂时放弃了。

Q96 : 不同的二叉搜索树 : 以 1 ... n 为节点组成的二叉搜索树有多少种?

1. 有一个公式: 考研的时候学过(记错了). 此处错误 这个公式是n个数出栈和入栈的顺序有多少种: c(2n,n)/(n+1)
2. 转化为线性dp问题. 即 123有几种,再加上一个4 又有几种.即我们知道1个数,2个数,3个数分别有多少种.那么4个数有多少种?
3. 4个数,选一个为root, 那么左侧0,右侧3,左侧1,右侧2. 左侧2,右侧1,左侧3,右侧0. 分为这些情况,全加一块即可.
4. 注意: 例如: 左侧分2,右侧分1, 左侧的种类m和右侧的种类n,是相乘的关系

```
        public int dfs(int i){
            if(dp[i] != 0) return dp[i];
            int sum = 0;
            // 根的划分。1...n，没有0
            for(int j = 1; j <= i/2; j++){
                sum = sum + dfs(j-1) * dfs(i - j);
            }
            sum *= 2;
            if(i % 2 != 0){
                int temp = dfs(i/2);
                sum = sum + temp * temp;
            }
            return dp[i] = sum;
        }
```

Q98：验证二叉搜索树：判定一个二叉搜索树是否合法.

1. 中序有序则合法. 使用全局的pre跟随指针.随着递归改变指向.作为前驱即可.

Q99：恢复二叉搜索树：二叉搜索树中两个结点被错误的交换,再不改变结构的情况下恢复. 不改变结构:即树的分支情况不改变.

1. 如果使用中序有序,很容易发现两个点的位置不对. 于是可以通过时间o(n)空间o(n)的算法来完成.

Q101：对称二叉树 从root处镜像对称

1. 这个题通过递归算写，还是蛮新的。 即同时对两颗树进行先序递归，

```
    l.val == r.val
    pre(l.left,r.right) pre(l.right, r.left)
```

Q106：从后序与中序遍历序列构造二叉树

1. 后序最后一个是root结点， 在中序中找到root结点的位置，划分为左右两侧序列。
2. 构造出根据左右划分，将后序序列进行划分
3. 构造出左右子树
4. 连接到root上
5. 返回root.

Q107：downToUp层次遍历： 先最后一层，向上。

1. 可以用dfs， dfs时根据层数，将元素加入到该层对应的列表中。

Q109 : 有序链表转换为平衡树

1. 先把链表变为 顺序表
2. 进行递归的分治算法， 让如果当前结点是奇数，那么左右平分， 如果是偶数，那么永远让左侧少右侧一个。

Q116 : 填充每个节点的下一个右侧节点指针 即树的结点多一个指针，是指向同层的下一个结点的。填充该指针。

Q117 :

1. prehead永远指向下一行第一个。
2. 让cur指向prehead, prehead再指向下一行第一个，然后pre指向每一行的dummy结点。
3. pre,cur伴随遍历该行，填充他们子节点的next指针。
4. 本质就是：使用上一层的next结点， 从逻辑结果来看，上一层的结构本质就是一个队列。

Q124 : 二叉树中的最大路径和

1. 后序递归，向上返回下边可向上拼接的最大路径即可。 路径可能从当前结点开始，也可能包含左子树，也可能包含右子树，三选一。

Q127 : 单词接龙： 两个词只有一个位置不同，那么就是相邻点。

0. 求两个单词在图中的距离。即经过多少次变化可以到达终点词
1. 层次遍历： 记录到达每个点的最短路径长度。
2. 优化建图， 如果我们拿出每两个结点，判断是否只差一个字符不同，那么效率太差了。

```
1. 假如 start: tot  end: dat     列表有单词 hot,dot

2. 那么我们通过虚拟结点把所有结点连在一起。

    tot- > *ot  t*t to*
    hot- > *ot  h*t ho*
    dot- > *ot  d*t do*
    dat- > *at  d*t da*
    那么就有 tot- > *ot- >   dot- >   d*t- >dat
    通过 map记录， *ot 的下标即可。

    public void addEdge(String word) {
        addWord(word);
        int id1 = wordId.get(word);
        char[] array = word.toCharArray();
        int length = array.length;
        for (int i = 0; i < length; ++i) {
            char tmp = array[i];
```

```
                array[i] = '*';
                String newWord = new String(array);
                // 此处可能不会添加结点。如果已经存在则不添加。
                addWord(newWord);
                // id2 就是 newWord的序号
                int id2 = wordId.get(newWord);
                edge.get(id1).add(id2);
                edge.get(id2).add(id1);
                array[i] = tmp;
            }
        }
```

Q173 : 二叉搜索树迭代器

1. 使用一个栈来模拟中序遍历的过程即可。

Q208 : 实现 Trie (前缀树)

```
            class Node{
                boolean exist;
                Node[] next = new Node[26]; // 使用链表，表示。// 对应字母的结点不为
null，则说明下边有结点。
                // 当exist = true时，说明该结点存在。
            }
```

Q222 : 完全二叉树的节点个数 完全二叉树：只有h层不满，且所有h层结点集中在树的左侧。

1. root结点的左子树高度，和右子树高度，可能相等，也可能不等。
2. 相等： 说明最后一层的叶子已经长到的右侧了。这个时候root的左子树是一颗满二叉树。可以直接算出结点数。
3. 不等： 说明左子树不一定是满的，但是 觉对 没有长到右侧。那么右侧就是满二叉树。

```
            int left = countLevel(root.left);// 求高度 求最左高度即可。因为是完全
二叉树。
            int right = countLevel(root.right);
            if(left == right){
                return countNodes(root.right) + (1 << left);
            }else{
                return countNodes(root.left) + (1 << right);
            }
```

Q230 : 二叉搜索树中第K小的元素

1. 利用中序有序。

Q236 : 二叉树的最近公共祖先

1. 很容易，从上往下，对于结点A，若 两个结点分别在A的左右两侧，那么A就是最近公共祖先。

Q331 : 验证二叉树的前序序列化 非空结点用数字表示，空结点用 #表示。

0. "9,3,4,#,#,1,#,#,2,#,6,#,#" 是一个合法的 前序序列。

1. #的个数 = 数字个数+1

2. 除了在最后一个位置。 其余位置 都有 数字个数 >= #个数

3. 问题转化： 出度：入度考虑。没有出度为1的结点。数字结点都是非叶子结点，且总是有两个子节点，子节点可能是数字，也可能是#

```
     1. # : 全为叶子结点。一颗没有出度为1的二叉树。的性质。
     2. 结点总数为n  数字结点为 n2  #结点为 n0
     3. 则 n2 + n0 = n
     4. 2*n2 + 1 = n = n2 + n0        所以 :  n2 + 1 = n0  就是说：这样的树总是满足:  数字结点+1 = #结点。
     5. 现在我们再考虑前序遍历的特点：总是先遍历 数字结点，再遍历叶子结点。 那么在遍历过程中。
     6. 数字结点+1应该总是 > #结点。   因为我们在遇到最后一个 # 结点时 才会有 n2 + 1 = n0 也就是说前边遍历时
     7. 总是有 n0 < n2 + 1  只有在i == len-1 时 才会有 n2 + 1 = n0 其余时候，必有n0 < n2 + 1

     只有当 n0太大时，才能发现这颗树有错误。
```

Q538 : 把二叉搜索树转换为累加树： 一颗二叉搜索树，更改树，树的结点存储所有大于等于该结点的所有值的和。

```
        4                              22
    1       6           -》         25      13
  0   2   5   7                  25    24  18   7
```

1. 对于 父，左，右三个结点， 在知道右子树的和后，才能计算 父结点的和，在 计算左结点的和前，要先计算父结点的和。

2. 所有递归循序应该是，右，父，左。 即把，中序遍历中， 左右结点的访问循序逆置。

3. 实际上，还是利用的搜索树的中序有序性， 只是我们是逆序递归的。才能在遍历中知道后边的和。

```
        值引用，结合 返回值。否者递归中val的变化，不会返回到本层。
        public int dfs(TreeNode root, int val//1){
            if(root != null){
                val = dfs(root.right, val); //2

                val = val + root.val;  //3
                root.val = val;

                val = dfs(root.left, val);
            }
            return val;
        }
        一个结点的和，来自于 其父节点的和，自身值，右子树的和。
```

JZ59：z字形遍历树

1. 可以使用双端队列，使用一个 flag来转向，使用null值作为一个分界符号。

```
//起初队列：   root,null
que.addLast(pRoot);
que.addLast(null);
// 首先从左侧出队。
boolean flag = true;
while(!que.isEmpty()){
    TreeNode top;
    if(flag) top = que.getFirst();
    // 左侧出队
    else top = que.getLast();
    // 右侧出队
    // 此处只是获取队头，没有出队
    if(top == null){
        // null 永远不出队列
        flag = !flag;
        // 切换方向
        ans.add(ls);
        ls = new ArrayList<Integer>();
        if(que.size() == 1) break;
        // 若只有一个 null，说明遍历完了。
        continue;
    }
    ls.add(top.val);
    // 左侧出队，则右侧入队。 先左子，再右子。
    if(flag) {
        que.removeFirst();
        if(top.left != null) que.addLast(top.left);
        if(top.right != null) que.addLast(top.right);
    } // 相反。
    else {
```

```
            que.removeLast();
            if(top.right != null) que.addFirst(top.right);
            if(top.left != null) que.addFirst(top.left);
        }
        // 构成了双端栈。从一侧出栈，在另一侧入栈。
    }
```